Оглавление

От авторов перевода

Это не первый наш опыт перевода книги, изданной много лет назад (Вирт, Гуткнехт). Но тем не менее мы взялись за перевод этой якобы старой книги, изданной 14 лет назад, потому что она не потеряла своей ценности и по сей день представляет интерес не только для профессионалов, но и для студентов, поскольку в ней подробно описывается, как путем незначительных усовершенствований давно и хорошо известных программистам структур данных можно добиться существенной эффективности работы с ними, что автор подкрепляет еще и доказательствами.

В сущности, книга представляет собой обзор различных способов модернизации давно известных структур данных с целью повышения их производительности. Поскольку в книге по большей части речь идет об оценке времени вычисления алгоритмов, то в таких случаях она подразумевается неявно. В тех же случаях, когда речь идет об оценке затрат памяти алгоритма, это явно оговаривается.

В целом книга представляет собой довольно полный обзор различных методов и приемов, позволяющих в той или иной степени ускорить работу с традиционными структурами данных, приводя необходимые обоснования (доказательства) их эффективности и во многих случаях – алгоритмы их реализации на языке программирования C. К сожалению, автор нередко приводит вместо программной реализации алгоритмов их словесное описание, что несколько затрудняет их понимание.

Книга может послужить хорошим справочником для всех тех, кто хочет понять, за счет каких приемов и методов можно ускорить работу с общеизвестными структурами данных.

Большинство так называемых автором «теорем» – это не столько теоремы, сколько выводы из предшествующих этим «теоремам» рассуждений. Книга насыщена такими «теоремами», но без доказательств. Обоснования этих «теорем» не всегда точны и убедительны. Те читатели, кто не интересуются ими, могут довериться автору и просто пропустить их. Ну а тем, кто хочет разобраться в них досконально, придется приложить определенные усилия или обратиться к многочисленным первоисточникам.

Конечно, эта книга адресована не столько программистам-любителям, сколько профессионалам, включая преподавателей и студентов, а также другим специалистам, интересующимся этой темой. Нужно сказать, что эта книга не для всех, а только для тех, кто хочет понять, с помощью каких методов и приемов можно достичь значительного ускорения работы алгоритмов с давно и многим известными структурами данных большого объема, включая базы данных.

Эта книга придется не всем по зубам, но мы приложили максимум усилий для того, чтобы ее перевод на русский язык хотя бы немного расширил круг ее читателей, и все найдут в ней немало познавательного и полезного.

О переводе

В главе 10 автор совершенно справедливо говорит о том, что термины и понятия в оригинальных работах не всегда удачны. Поэтому при переводе мы, конечно же, старались придерживаться авторской терминологии и тех оригинальных работ, на которые он ссылается, но изредка позволяли себе употреблять более понятные, с нашей точки зрения, русскоязычные термины. Что из этого получилось, судить читателю.

Несмотря на то что книга может заинтересовать довольно узкий круг читателей, мы надеемся, что наш перевод несколько расширит этот круг. Оригинал книги представляет собой скорее конспект лекций, чем детально продуманную монографию. Мы постарались превратить ее в полноценную монографию и, надеемся, интересную книгу для более широкого круга читателей – от студентов и препо-

давателей до профессиональных программистов (специалистов), которые, как мы думаем, не разочаруются в ней.

В предисловии к книге автор пишет, что она родилась из его набросков лекций по данной теме. И к сожалению, этот набросочный стиль изложения материала отчасти сохранился и в самой книге. Поэтому при переводе книги нам пришлось немало потрудиться, чтобы превратить авторские наброски в полноценную монографию, качество которой оценит наш читатель.

Несколько слов о том, что красной нитью проходит по всей книге, – это оценка пространственно-временно́й сложности алгоритмов. Для них обычно используются обозначения: O – верхняя асимптотическая оценка сложности для худшего случая, Ω – нижняя асимптотическая оценка для худшего случая и Θ – жесткая асимптотическая оценка, когда сложность алгоритма для худшего случая не может выходить за пределы определяемых ею границ. Кроме того, нередко в книге идет речь об амортизированной оценке вычислений, которую мы называли средней или усредненной. Чаще всего речь в книге идет именно об асимптотической оценке временно́й сложности алгоритмов для худшего случая и лишь изредка оценивается нижняя, жесткая, средняя и пространственная сложность алгоритмов.

При переводе книги мы позволили себе ради ее улучшения, помимо прочих незначительных вольностей, пронумеровать все непронумерованные авторские рисунки, а также обширный список публикаций.

Терминология

Об оценке сложности алгоритмов.

Сложность алгоритма = оценка стоимости или цены, затраты…

Временна́я и пространственная (= затраты основной памяти).

Временная сложность = количество выполняемых операций или время их выполнения.

Пространственная сложность = затраты (объем) памяти для выполнения операций.

Чаще всего автор имеет в виду временну́ю сложность.

hieght balanced tree – выровненное по высоте дерево = равновысокое дерево.

weight balanced tree – уравновешенное дерево = уравновешенное (равновесное) дерево.

rebalancing – выравнивание.

amortized – амортизированная (смягченная, усредненная или средняя) для худшего случая оценка сложности вычислений.

Формулы

Некоторые несложные «многоэтажные» математические выражения, формулы или обычные дроби мы иногда заменяли строчной записью, конечно же, без потери их смысла.

Так, например,

$$\frac{-1}{(\alpha \log \alpha + (1-\alpha)\log(1-\alpha))\log n}$$

мы заменили на

$$-1/(\alpha \log \alpha + (1-\alpha)\log(1-\alpha))\log n.$$

Е. В. Борисов, С. В. Чугунов,
Март, 2023

Предисловие

Эта книга – учебник для слушателей курсов по структурам данных. Структура данных – это метод[1] представления данных для выполнения над ними некоторого набора операций. Классический пример структуры данных – это множество распознаваемых по значению ключа элементов в виде пар (*ключ, элемент*), которые можно добавлять, удалять или искать в этом множестве по заданному ключу. Структура, поддерживающая такие операции, называется словарем (dictionary). Словари могут быть реализованы разными способами, с разной степенью сложности и разными дополнительными операциями. Во многих первоисточниках было предложено и исследовано множество видов словарей, и некоторые из них[2] мы рассмотрим в этой книге.

Вообще говоря, структура данных – это своего рода набор высокоуровневых операций некоторой виртуальной машины: если алгоритм многократно выполняет некоторые операции, можно определить, какие из них выполняются чаще всего и как их лучше реализовать. Таким образом, основная задача структур данных – как реализовать набор операций над ними, предполагаемый результат которых нам известен.

Сегодня нет недостатка в книгах, в названии которых есть словосочетание «*структуры данных*», но они лишь поверхностно касаются этой темы, затрагивая только простейшие структуры – стек, очередь и некоторые виды выровненных деревьев поиска с изрядным количеством пафоса (handwaving). Серьезный интерес к структурам данных начали проявлять в 1970-х годах, а в первой половине 1980-х почти в каждом номере журнала *Communications ACM* была статья о них. Структуры данных стали центральной темой, отдельным пунктом в *классификаторе информатики*[3] и стандартной частью учебных программ по информатике[4]. С выходом книги Вирта «*Структуры данных + Алгоритмы = Программы*» термин «*алгоритмы и структуры данных*» стал общепринятым в учебниках по этой теме. Правда, единственная монография Овермарса 1983 года [450], посвященная именно алгоритмическому аспекту структур данных, до сих пор находится в печати; это своего рода рекорд для серии книг *LNCS* (*Lecture Notes in Computer Science*). Структурам данных уделяется внимание во многих приложениях и прежде всего в базах данных как индексным структурам. В том же контексте в монографиях Самета [482, 483] были изучены структуры для

[1] Эта книга не об объектно ориентированном программировании, поэтому термины «метод» и «объект» употребляются в ней в самом обычном смысле.

[2] Наиболее интересные и значимые. – *Прим. перев.*

[3] Классификационный код Е.1. «Структуры данных». К сожалению, Классификатор информатики слишком груб, чтобы быть полезным.

[4] **ABET** (**A**ccreditation **B**oard for **E**ngineering and **T**echnology, Inc. – *Прим. перев.*) по-прежнему причисляет их к одной из пяти основных тем: алгоритмы, структуры данных, разработка ПО, языки программирования и архитектура компьютеров.

геометрических данных; подобные структуры данных были исследованы Лангетепе и Заманом [346] для компьютерной графики. В последнее время были подробно изучены строковые структуры данных, востребованные в первую очередь приложениями биоинформатики. Не иссякает поток публикаций по теории структур данных для вычислительной геометрии и комбинаторики. Однако во многих учебниках структуры данных рассматриваются только как примеры для объектно ориентированного программирования, исключая их важнейший алгоритмический аспект – как реализовать нетривиальные структуры данных, не выходя при этом за пределы худших оценок вычислительной сложности. Цель этой книги – сосредоточиться на структуре данных как основе любого алгоритма. Недавно изданный *Справочник по структурам данных* [411] – это еще один шаг в этом же направлении.

В книге приводится реальный код для многих обсуждаемых в ней структур данных и довольно подробная информация о большинстве других структур данных, реализация которых в книге не дается. Многие учебники избегают таких подробностей, что является одной из причин, по которой структуры данных не используются там, где они должны быть. Выбор рассматриваемых в этой книге структур данных почти всюду ограничивается только теми структурами, которые вписываются в модель так называемой *ссылочной машины* (pointer-machine)[5], за исключением хеш-таблиц, которые рассматриваются из-за их практической важности. Все коды книги – сугубо иллюстративные и не предназначены для использования в качестве готовых модулей; их правильность не гарантируется. Большинство кодов доступны на моей домашней странице с минимальными возможностями их проверки.

Эта книга начиналась с набросков курса, который я читал в зимнем семестре 2000 года в Берлинском свободном университете (Free University of Berlin). Я благодарю Христиана Кнауэра, который был тогда моим ассистентом; мы с ним многому научились. Затем я представил этот курс в осенних семестрах 2004–2007 годов аспирантам Нью-Йоркского городского колледжа (City College of New York), а в июле 2006 года взял его за основу для летней школы по структурам данных в Корейском институте передовых технологий (Korean Advanced Institute of Science and Technology – KAIST). Работа над книгой была закончена в ноябре 2007 года.

Благодарю Эмили Войтек и Гюнтера Роте за выявленные в моих кодах ошибки, Отфрида Чонга – за организацию летней школы в KAIST, а также самих участников летней школы – за выявленные ими мои прочие ошибки. Благодарю Христиана Кнауэра и Хельмута Брасса за литературу из превосходных математических библиотек Берлинского свободного университета и Брауншвейгского технического университета, а также Яноша Паха за доступ к онлайн-журналам Института Куранта. Эта книга

[5] Один из вариантов абстрактной машины с косвенной адресацией (Random Access Machine, сокращенно – RAM), предназначенной для теоретической оценки сложности алгоритмов. – *Прим. перев.*

была бы невозможна без доступа к хорошим библиотекам, и я старался цитировать только те источники, которые сам просматривал.

Данная книга не поддерживалась ни одним грантовым агентством.

Основные понятия

Структура данных моделирует некоторый абстрактный объект, обладающий набором операций, которые обычно можно разделить на:

- операции создания и удаления;
- операции обновления;
- операции обращения к данным (запросы).

В случае словаря мы создаем или удаляем объект, обновляем его, добавляя или удаляя его элементы, или запрашиваем его элемент путем обращения к нему. После создания объекта он изменяется посредством операций обновления. Операции обращения не изменяют сам абстрактный объект, но могут изменить вид объекта внутри структуры данных. Это называется *адаптивной* (adaptive) структурой данных, которая может перестраиваться после каждого обращения к ней с целью ускорения последующих обращений к ее данным.

Допускающие подобные обновления и обращения структуры данных называются *динамическими* (dynamic). Правда, есть более простые структуры, которые создаются только один раз и допускают обращения к своим элементам без их изменения; они называются *статическими* (static). Динамические структуры данных, как более общие, предпочтительнее, однако мы не должны пренебрегать и статическими структурами, так как они являются строительными блоками для динамических структур, хотя для некоторых довольно сложных объектов, с которыми мы столкнемся, динамическая структура вообще неизвестна.

Наша цель – определить и реализовать структуру данных так, чтобы она обеспечивала максимальную скорость работы с заданным абстрактным объектом. Размер структуры – еще один ее качественный показатель, но его влияние на скорость обычно незначительно. Для оценки скорости нужна некая мера. Обычно ею является размер самого объекта, а не его представление. Отметим, что в результате многократного применения операции обновления (добавления/удаления) размер объекта может даже уменьшиться. Наиболее общая мера сложности вычислений – это так называемая *худшая* (*worst-case*) сложность[6]. Таким образом, сложность операции с заданной структурой данных оценивается величиной $O(f(n))$, если в любом состоянии структуры данных после применения ряда операций обновления, приведших объект к размеру n, эта операция требует времени,

[6] То есть *асимптотическая верхняя граница вычислительной сложности алгоритма.* Далее в книге автор называет эту оценку *временем* выполнения алгоритма, что не вполне правильно. – *Прим. перев.*

не превышающего $Cf(n)$ для некоторой константы C. Альтернативная, но более мягкая мера – это *средняя* (*amortized*) сложность[7]. Операция обновления имеет среднюю сложность $O(f(n))$, если существует такая функция $g(n)$, что любая последовательность из m таких операций, не увеличивающих размер n самого объекта, требует времени не более $g(n) + mCf(n)$, поэтому при многократном применении такой операции сложность вычислений в среднем не превысит $Cf(n)$.

Некоторые структуры данных можно отнести к *случайным* (randomized). Обращение к одному и тому же объекту выполняется неким случайным образом, и одни и те же действия над ним не всегда приводят к одинаковой последовательности шагов. В этом случае оценивается *ожидаемая* (*expected*) сложность операции; она тоже случайна и потому относится к худшей для всех объектов того же размера с теми же операциями над ними.

В некоторых случаях худшая оценка сложности вычислений $O(f(n))$ может ухудшаться из-за большого размера результата (выхода) операции, влияющего на ее сложность. В таких случаях имеет значение еще и чувствительность алгоритма к выходу, размер которого влияет на скорость алгоритма. Операция имеет *чувствительную к выходу* (*output-sensitive*) сложность $O(f(n) + k)$, если для объекта размера n и выхода размера k она требует времени не более $C(f(n) + k)$.

Для динамических структур данных время создания нового объекта обычно постоянно, поэтому нас будут интересовать главным образом сложность операций обновления объекта структуры и обращения к нему. Время удаления структуры размера n почти всегда равно $O(n)$. Для статических структур данных объект размера n создается сразу и не обновляется, поэтому в этом случае нас будет интересовать только время его создания (предварительной обработки) и время обращения к нему.

В этой книге $\log_a n$ – это логарифм по основанию a, но если основание логарифма не указано, то оно равно 2.

Для интервалов используется стиль записи Бурбаки: $[a, b]$ и $]a, b[$ – соответственно закрытый и открытый интервалы от a до b, а $]a, b]$ и $[a, b[$ – полуоткрытые интервалы с исключением, соответственно, начала и конца.

Подобно приведенному выше обозначению $O(\cdot)$ для асимптотической верхней границы вычислительной сложности, мы будем использовать также обозначения $\Omega(\cdot)$ – для нижней границы и $\Theta(\cdot)$ – для жесткой (верхней и нижней) границы. Неотрицательная функция f принадлежит $O(g(n))$ или $\Omega(g(n))$, если для некоторого положительного C и любых достаточно больших n выполняется условие $f(n) \leqslant Cg(n)$ или $f(n) \geqslant Cg(n)$ соответственно. Таким образом, функция f принадлежит $\Theta(g(n))$, если она принадлежит одно-

[7] Обычно это некая усредненная оценка сложности выполнения цепочки операций над структурой данных, когда оценивается их *средняя* сложность в худшем случае. Она применяется, когда наиболее сложные операции выполняются довольно редко, но средняя сложность всей цепочки операций оказывается вполне приемлемой. Отметим, что средняя оценка не является вероятностной (ожидаемой). Это всего лишь средняя оценка $a = (\sum_{i=1}^{n} t_i)/n$, где t_1, t_2, ..., t_n – сложность (или времена) выполнения в худшем случае цепочки из n операций над структурой данных. – *Прим. перев.*

временно $O(g(n))$ и $\Omega(g(n))$. Здесь «достаточно большие n» – это те значения, для которых $g(n)$ определена и положительна.

Примеры кода

Примеры кода в этой книге приведены в стандарте языка C, но они будут вполне понятны всем, кто знаком с любым другим императивным языком программирования.

В примерах кода символ «=» обозначает присваивание, а символ «==» – проверку на равенство. За пределами кода символ «=» имеет обычный математический смысл.

Логические операции отрицания, конъюнкции и дизъюнкции обозначаются в коде соответственно символами «!», «&&» и «||», а символ «%» обозначает операцию деления по модулю.

Разыменование указателя (ссылки) выполняется операцией «*»: если `pt` – указатель на область памяти, то `*pt` – это сама область памяти. Указатели имеют тип, определяющий способ интерпретации содержимого этой области памяти. Таким образом, при объявлении указателя объявляется и тип области памяти, на которую он ссылается. Так, например, «`int *pt;`» объявляет указатель `pt` на область памяти типа `int`. Указатели – это переменные, которые допускают операции присваивания и добавления к ним целого числа (указательная арифметика). Если `pt` указывает на область памяти определенного типа, то `pt + 1` указывает на следующую область памяти такого же типа. Иными словами, каждый объявленный указатель трактует всю память как большой массив однотипных элементов. `NULL` – это указатель, который не указывает ни на какую область памяти, и его можно использовать в качестве особого значения в операциях сравнения.

Структура – это определяемый пользователем тип данных, представляющий собой последовательность именованных элементов (полей) любого типа. Элемент структуры тоже может быть структурой, но отличной от типа самой структуры, иначе структура становится рекурсивной, то есть бесконечной. Но зато ее элементом вполне может быть указатель на объект того же типа, что и сама структура. Именно такие структуры обычно бывают объектами в теории структур данных. Полям структур можно присваивать значения и использовать их подобно любым другим переменным. Если, например, переменная `z` определена как структура следующего типа `C`:

```
typedef struct { float x; float y; } C,
```

то ее элементы `z.x` и `z.y` имеют тип `float`. Если переменная `zpt` объявлена как указатель на объект типа `C` (`C *zpt;`), то она может ссылаться на `(*zpt).x` и `(*zpt).y`. Для такой часто используемой комбинации операций разыменования указателя и обращения к элементу структуры в языке C существует альтернативная операция `zpt->x` и `zpt->y`. Эти операции эквивалентны, но последняя предпочтительнее, поскольку снимает проблему приоритета

операций: разыменование имеет более низкий, чем обращение к элементу структуры, приоритет, поэтому `(*zpt).x` – это не то же самое, что `*zpt.x`.

Во всех функциях мы избегаем рекурсивных вызовов, хотя в некоторых случаях они могут существенно упростить код. Однако значительные издержки рекурсивных вызовов функций вступают в противоречие с нашей целью достижения наивысшей скорости работы со структурами данных. При этом мы не противники рекурсивных функций, поскольку хороший компилятор сам может избавиться от них, используя встроенные функции или макросы.

Кроме того, в тексте книги объект данных нередко будет отождествляться со ссылкой на него.

Глава 1

Элементарные структуры

Элементарные структуры данных *стек* и *очередь* обычно изучаются в курсе «Программирование 2». Их объединяет иногда упоминаемая, но редко используемая на практике *двусторонняя очередь*. Стек и очередь – основные структуры данных, именно поэтому они будут подробно обсуждаться и использоваться для иллюстрации отдельных моментов реализации других структур данных.

1.1. Стек

Стек – простейшая из всех структур с очевидной интерпретацией: добавить (втолкнуть) объект в стек и затем удалить (вытолкнуть) его из стека с доступом только к верхнему объекту (вершине). По этой причине его иногда называют памятью типа LIFO (Last In, First Out – последним вошел, первым вышел)[1]. В программировании стеки встречаются всюду, где есть вложенные блоки, локальные переменные, рекурсия или перебор с возвратами (backtracking). Типичные примеры программ с использованием стеков – это вычисление скобочных арифметических выражений с различными приоритетами операций или поиск пути в лабиринте методом перебора с возвратами.

Стек должен поддерживать, по крайней мере, следующие операции:

- `push(obj)` – добавить (втолкнуть) объект `obj` в вершину стека;
- `pop()` – удалить (вытолкнуть) объект из вершины стека;
- `stack_empty()` – проверить, пуст ли стек.

Реализация этих операций должна, конечно же, обеспечивать правильные результаты, поэтому необходимо как-то определить правильное поведение стека. Одним из способов может быть алгебраическое определение операций и возвращаемых ими значений. Для простых структур, подобных стеку, это возможно, но для понимания структуры такое определение не очень полезно. Вместо него можно определить каноническую реализацию на некой абстрактной машине с бесконечной памятью, которая выдает

[1] Типичный пример LIFO – пистолетная обойма с патронами. – *Прим. перев.*

правильные результаты для всех правильных цепочек операций (исключая операцию удаления из пустого стека). Если элементы стека имеют тип `item_t`, то такая реализация может выглядеть следующим образом:

```
int i=0;
item_t stack[∞];
int stack_empty(void)
{ return(i == 0);
}

void push( item_t x)
{ stack[i++] = x;
}

item_t pop(void)
{ return(stack[--i]);
}
```

Операции со стеком реализованы правильно, но в них есть одна проблема – это массив бесконечного размера, в котором всякая последовательность операций всегда будет правильной. Поэтому более реалистичной может быть следующая версия:

```
int i=0;
item_t stack[MAXSIZE];

int stack_empty(void)
{ return(i == 0);
}

int push(item_t x)
{ if( i < MAXSIZE )
     { stack[i++] = x;
       return(0);
     }
     else
       return(-1);
}

item_t pop(void)
{ return(stack[--i]);
}
```

Введенное ограничение максимального размера стека ограничивает правильное поведение стека, поэтому это не совсем то, что нам хотелось бы иметь. Но зато в этой реализации операция `push` возвращает сообще-

ние об ошибке (–1) при переполнении стека. Главный недостаток реализации структур данных в массиве заключается в том, что массивы имеют фиксированный размер, который должен быть задан заранее вне зависимости от истинного количества элементов структуры. Для устранения этого недостатка существует систематический способ, рассматриваемый в разделе 1.5, но предпочтение обычно отдается решению с динамически выделяемой памятью.

В этой реализации ошибка выдается только при *переполнении* (overflow) стека, но не при его *опустошении* (underflow), поскольку переполнение стека – это ошибка, порожденная самой структурой, что невозможно в идеальной реализации, тогда как опустошение стека – это ошибка неправильного использования структуры и, следовательно, результат работы программы, считающей стек «черным ящиком». Ее функция pop возвращает вершину стека, но если нужно «поймать» ошибку опустошения стека, то она должна возвращать еще и признак ошибки. Последняя претензия к первоначальной версии реализации стека состоит в том, что в одной и той же программе может понадобиться не один, а несколько стеков, которые придется создавать динамически. Поэтому нам нужны дополнительные операции по созданию и удалению стеков, а каждая операция со стеком должна знать, с каким именно стеком она работает. Одним из возможных вариантов реализации стека может быть следующий:

```
typedef struct {item_t *base; item_t *top; int size;} stack_t;

stack_t *create_stack(int size)
{ stack_t *st;
  st = (stack_t *) malloc(sizeof(stack_t));
  st->base = (item_t *) malloc(size * sizeof(item_t));
  st->size = size;
  st->top = st->base;
  return(st);
}

int stack_empty(stack_t *st)
{ return(st->base == st->top);
}

int push( item_t x, stack_t *st)
{ if (st->top < st->base + st->size)
  { *(st->top) = x;
      st->top += 1;
      return( 0 );
  }
  else
     return( -1 );
```

```
}

item_t pop(stack_t *st)
{ st->top -= 1;
  return(*(st->top));
}

item_t top_element(stack_t *st)
{ return(*(st->top -1));
}

void remove_stack(stack_t *st)
{ free(st->base);
    free(st);
}
```

Сюда мы включаем только некоторые проверки безопасности и исключаем другие. В общем, наша политика безопасности состоит в том, чтобы включать только те проверки, которые, в отличие от идеального стека, выявляют ошибки, вызванные ограничениями этой реализации при правильном использовании стека и неограниченном объеме памяти базовой операционной системы. Кроме того, мы добавили еще одну полезную операцию, которая просто возвращает вершину стека без ее удаления.

Часто более предпочтительной реализацией стека является структура с динамически выделяемой памятью в виде связного списка, когда добавление и удаление элемента выполняется в начале списка. Преимущество такого решения в том, что у такой структуры нет фиксированного размера, поэтому (считая память компьютера неограниченной) к ней всегда можно добавить новый элемент без проверки на ошибку переполнения стека. Она так же проста, как и структура в массиве, при наличии функций `get_node` и `return_node`, реализация которых будет приведена в разделе 1.4.

```
typedef struct st_t { item_t item; struct st_t *next; } stack_t;

stack_t *create_stack(void)
{ stack_t *st;
  st = get_node();
  st->next = NULL;
  return(st);
}

int stack_empty(stack_t *st)
{ return(st->next == NULL);
}

void push(item_t x, stack_t *st)
```

```
{  stack_t *tmp;
   tmp = get_node();
   tmp->item = x;
   tmp->next = st->next;
   st->next = tmp;
}

item_t pop(stack_t *st)
{ stack_t *tmp; item_t tmp_item;
  tmp = st->next;
  st->next = tmp->next;
  tmp_item = tmp->item;
  return_node(tmp);
  return(tmp_item);
}

item_t top_element(stack_t *st)
{ return(st->next->item);
}

void remove_stack(stack_t *st)
{ stack_t *tmp;
  do
  { tmp = st->next;
    return_node(st);
    st = tmp;
  }
  while( tmp != NULL );
}
```

Обратите внимание, что связный список начинается с пустого элемента (заголовка), поэтому пустой стек – это список из одного заголовка, а вершина стека – это уже следующий элемент списка. Это нужно для того, чтобы возвращаемый функцией `create_stack` и используемый во всех операциях указатель стека не изменялся ими. Поэтому указатель вершины стека не может быть его идентификатором. Поскольку извлекаемые из списка элементы становятся недоступными, нам нужны временные копии их значений в `pop` и `remove_stack`. Операция `remove_stack` должна возвращать все оставшиеся в стеке элементы. Однако нет оснований полагать, что удаляться будут только пустые стеки, поэтому при неудачной попытке удалить оставшиеся элементы не исключена утечка памяти.

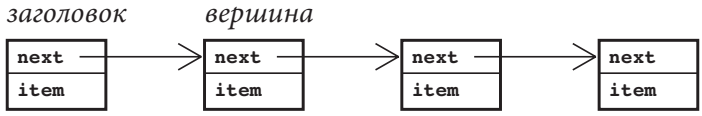

Рис. 1.1. Стек в виде списка из трех элементов

Реализация структуры с динамически выделяемой памятью более элегантна. Ей не грозит переполнение стека, и ей нужна только фактически используемая память, а не большие массивы, предельный размер которых определяется программистом. Один из недостатков такой реализации – возможное снижение производительности: разыменование указателя по времени не больше, чем увеличение индекса, однако адрес памяти, куда ссылается указатель, может находиться в любом месте памяти, тогда как следующий элемент массива находится вслед за предыдущим. Таким образом, структуры в массиве обычно очень хорошо работают с кешем, тогда как структуры с динамически выделяемой памятью могут создавать в кеше много пустот. Поэтому если мы совершенно уверены в максимально возможном размере стека (например, потому что его размер – всего лишь логарифм размера входных данных), то предпочтительна его реализация в массиве.

Если же нужно объединить достоинства обеих реализаций, можно использовать связный список блоков (массивов), по заполнении каждого из которых к списку добавляется новый блок. Такая реализация может выглядеть следующим образом:

```
typedef struct st_t { item_t  *base;
                      item_t   *top;
                       int     size;
                struct st_t  *previous; } stack_t;

stack_t *create_stack(int size)
{ stack_t *st;
  st = (stack_t *) malloc(sizeof(stack_t));
  st->base = (item_t *) malloc(size * sizeof(item_t));
  st->size = size;
  st->top = st->base;
  st->previous = NULL;
  return(st);
}

int stack_empty(stack_t *st)
{ return( st->base == st->top && st->previous == NULL);
}

void push( item_t x, stack_t *st)
{ if( st->top < st->base + st->size )
   { *(st->top) = x;
       st->top += 1;
  }
  else
  { stack_t *new;
```

```
    new = (stack_t *) malloc(sizeof(stack_t));
    new->base = st->base;
    new->top = st->top;
    new->size = st->size;
    new->previous = st->previous;
    st->previous = new;
    st->base = (item_t *) malloc(st->size * sizeof(item_t));
    st->top = st->base + 1;
    *(st->base) = x;
  }
}

item_t pop(stack_t *st)
{ if( st->top == st->base )
  { stack_t *old;
    old = st->previous;
    st->previous = old->previous;
    free(st->base);
    st->base = old->base;
    st->top = old->top;
    st->size = old->size;
    free(old);
  }
  st->top -= 1;
  return(*(st->top));
}

item_t top_element(stack_t *st)
{ if( st->top == st->base )
     return(*(st->previous->top - 1));
  else
     return(*(st->top - 1));
}

void remove_stack(stack_t *st)
{ stack_t *tmp;
  do
  { tmp = st->previous;
    free(st->base);
    free(st);
    st = tmp;
  }
  while( st != NULL );
}
```

Согласно нашей классификации операции `push` и `pop` обновляют структуру, тогда как `stack_empty` и `top_element` просто обращаются к ней. Очевидно, что реализация в массиве позволяет выполнять все действия за постоянное время, поскольку они включают только постоянное число операций. Реализация в виде связного списка предполагает при выполнении операций `push` и `pop` разовые вызовы внешних функций `get_node` и `return_node`, поэтому они требуют постоянного времени, но лишь в предположении, что сами эти функции имеют фиксированное время выполнения. Реализацию с динамическим созданием элементов мы обсудим в разделе 1.4, но можем здесь (и во всех остальных структурах) допустить, что время их выполнения постоянно. Для связного списка блоков вместо создания промежуточного слоя, как описано в разделе 1.4, используются стандартные операции управления памятью `malloc` и `free`, которые выделяют и освобождают большие области памяти. Обычно считается, что выделение и освобождение памяти – это операции с постоянным временем выполнения, но (в особенности для операции `free`) здесь возникают нетривиальные проблемы, поэтому следует избегать их частого употребления. В случае списка блоков это может произойти, например, при наличии большого количества операций `push` / `pop`, которые могут выводить за пределы выделенного блока. Так что незначительные преимущества связного списка блоков, видимо, не стоят лишних проблем.

В операции `create_stack` выполняется только одно выделение памяти, и потому время ее выполнения при любой реализации должно быть постоянным. А вот операция `remove_stack`, очевидно, не имеет постоянного времени выполнения, поскольку должна удалять потенциально большую структуру: если в стеке n элементов, то время его удаления – $O(n)$.

1.2. Очередь

Очередь – почти такая же простая структура, как стек; элементы в ней хранятся так же, но в отличие от стека сначала извлекаются те, что были помещены в нее первыми, то есть это память типа FIFO (First In, First Out – первым вошел, первым вышел). Очереди полезны в задачах с циклической обработкой данных. Кроме того, они являются центральной структурой при поиске в ширину (Breadth-First Search, BFS). Поиск в ширину и поиск в глубину (Depth-First Search, DFS) различаются, по сути, только тем, что для хранения очередного объекта BFS использует очередь, а DFS – стек.

Очередь должна поддерживать, по крайней мере, следующие операции:

- `enqueue(obj)` – добавить объект `obj` в конец очереди;
- `dequeue()` – вернуть 1-й объект очереди с удалением его из очереди;
- `queue_empty()` – проверить, пуста ли очередь.

Разница между стеком и очередью, делающая ее чуть сложнее, – в том, что изменения происходят на обоих ее концах: на одном конце – добавле-

ния, на другом – удаления. Если очередь реализуется в массиве, то используемый ею отрезок массива как бы перемещается внутри него. При бесконечном массиве ее реализация не представляла бы никакой проблемы и выглядела бы так:

```
int lower=0; int upper=0;
item_t queue[∞];
int queue_empty(void)
{ return(lower == upper);
}

void enqueue(item_t x)
{ queue[upper++] = x;
}

item_t dequeue(void)
{ return(queue[lower++]);
}
```

На практике очередь в массиве ограниченной длины реализуется в виде кольца, когда индекс вычисляется по модулю длины массива. Тогда ее реализация могла бы выглядеть так:

```
typedef struct { item_t *base;
                 int    front;
                 int     rear;
                 int     size; } queue_t;

queue_t *create_queue(int size)
{ queue_t *qu;
  qu = (queue_t *) malloc( sizeof(queue_t) );
  qu->base = (item_t *) malloc( size *sizeof(item_t) );
  qu->size = size;
  qu->front = qu->rear = 0;
  return(qu);
}

int queue_empty(queue_t *qu)
{ return(qu->front == qu->rear);
}

int enqueue(item_t x, queue_t *qu)
{ if( qu->front != ((qu->rear +2)% qu->size) )
  { qu->base[qu->rear] = x;
    qu->rear = ((qu->rear+1)%qu->size);
    return(0);
```

```
  }
  else
    return(-1);
}

item_t dequeue(queue_t *qu)
{ int tmp;
  tmp = qu->front;
  qu->front = ((qu->front +1)%qu->size);
  return(qu->base[tmp]);
}

item_t front_element(queue_t *qu)
{ return(qu->base[qu->front]);
}

void remove_queue(queue_t *qu)
{ free(qu->base);
  free(qu);
}
```

И здесь опять виден главный недостаток всякой структуры, реализованной в массиве, – его фиксированный размер. А это значит, что возможны ошибки переполнения и неправильная реализация структуры из-за его ограниченности. Кроме того, в этом случае всегда задается предполагаемый максимальный размер массива, который может оказаться невостребованным. Более предпочтительна альтернативная структура в виде связного списка с динамически выделяемой памятью; очевидной ее реализацией может быть такая:

```
typedef struct qu_n_t { item_t item; struct qu_n_t *next; } qu_node_t;
typedef struct { qu_node_t *remove; qu_node_t *insert; } queue_t;

queue_t *create_queue()
{ queue_t *qu;
  qu = (queue_t *) malloc(sizeof(queue_t));
  qu->remove = qu->insert = NULL;
  return(qu);
}

int queue_empty(queue_t *qu)
{ return(qu->insert == NULL);
}

void enqueue(item_t x, queue_t *qu)
```

```
{ qu_node_t *tmp;
  tmp = t_node();
  tmp->item = x;
  tmp->next = NULL; /* концевой маркер */
  if( qu->insert != NULL ) /* очередь не пуста */
  { qu->insert->next = tmp;
    qu->insert = tmp;
  }
  else /* добавить в пустую очередь */
  { qu->remove = qu->insert = tmp;
  }
}

item_t dequeue(queue_t *qu)
{ qu_node_t *tmp; item_t tmp_item;
  tmp = qu->remove;
  tmp_item = tmp->item;
  qu->remove = tmp->next;
  if( qu->remove == NULL ) /* достигнут конец */
      qu->insert = NULL; /* опустошить очередь */
  return_node(tmp);
  return(tmp_item);
}

item_t front_element(queue_t *qu)
{ return(qu->remove->item);
}

void remove_queue(queue_t *qu)
{ qu_node_t *tmp;
  while( qu->remove != NULL )
  { tmp = qu->remove;
    qu->remove = tmp->next;
    return_node(tmp);
  }
  free(qu);
}
```

Как и для всех динамических структур, мы снова считаем, что нам доступны операции `get_node` и `return_node`; они всегда работают правильно и выполняются с постоянным временем. Поскольку элементы удаляются из начала очереди, указатели в связном списке направлены от его начала к концу, куда добавляются новые элементы. Есть два эстетических недостатка этой очевидной реализации: требуется особая, отличная от остальных элементов списка точка входа и иная реализация операций с пустой оче-

редью. При добавлении в пустую очередь и удалении из нее последнего и единственного элемента придется изменить указатели и добавления, и удаления, а для всех остальных операций будет изменяться лишь один из этих указателей.

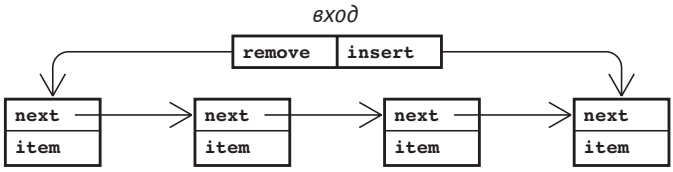

Рис. 1.2. Очередь в виде списка из четырех элементов

Первый недостаток можно устранить закольцовыванием списка, сделав его циклическим, когда указатель последнего элемента очереди ссылается на первый. В этом случае можно обойтись без указателя удаления, потому что следующий элемент точки добавления ссылается на точку удаления. Таким образом, начальный элемент очереди нуждается только в одном указателе и потому имеет тот же тип, что и все остальные элементы очереди.

Второй недостаток можно устранить добавлением в этот циклический список элемента-заголовка – между концом добавления и концом удаления. Точка входа по-прежнему указывает на конец добавления или, в случае пустого списка, на заголовок. Тогда, по крайней мере при добавлении, пустой список больше не будет особым случаем.

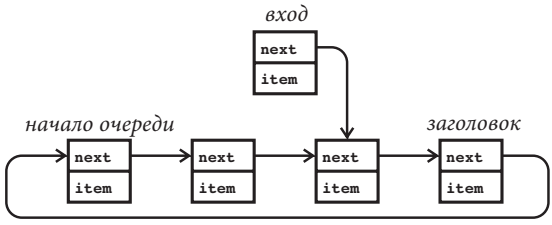

Рис. 1.3. Очередь в виде циклического списка из трех элементов

Таким образом, реализация очереди в виде циклического списка может выглядеть так:

```
typedef struct qu_t { item_t item; struct qu_t *next; } queue_t;

queue_t *create_queue()
{ queue_t *entrypoint, *placeholder;
  entrypoint = (queue_t *) malloc(sizeof(queue_t));
  placeholder = (queue_t *) malloc(sizeof(queue_t));
  entrypoint->next = placeholder;
  placeholder->next = placeholder; return(entrypoint);
}

int queue_empty(queue_t *qu)
```

```
{ return(qu->next == qu->next->next);
}

void enqueue(item_t x, queue_t *qu)
{ queue_t *tmp, *new;
  new = get_node();
  new->item = x;
  tmp = qu->next;
  qu->next = new;
  new->next = tmp->next;
  tmp->next = new;
}

item_t dequeue(queue_t *qu)
{ queue_t *tmp;
  item_t tmp_item;
  tmp = qu->next->next->next;
  qu->next->next->next = tmp->next;
  if( tmp == qu->next )
     qu->next = tmp->next;
  tmp_item = tmp->item; return_node(tmp);
  return( tmp_item );
}

item_t front_element(queue_t *qu)
{ return(qu->next->next->next->item);
}

void remove_queue(queue_t *qu)
{ queue_t *tmp;
  tmp = qu->next->next;
  while( tmp != qu->next )
  { qu->next->next = tmp->next; return_node(tmp);
    tmp = qu->next->next;
  }
  return_node(qu->next);
  return_node(qu);
}
```

Еще очередь можно реализовать в виде двусвязного списка, который почти не отличается от предыдущего, но требует двух указателей на каждый элемент. Минимизация количества указателей – это чисто эстетический критерий, определяемый скорее объемом выполняемой на каждом шаге работы с сохранением связности структуры, нежели объемом, необходимым для структуры памяти.

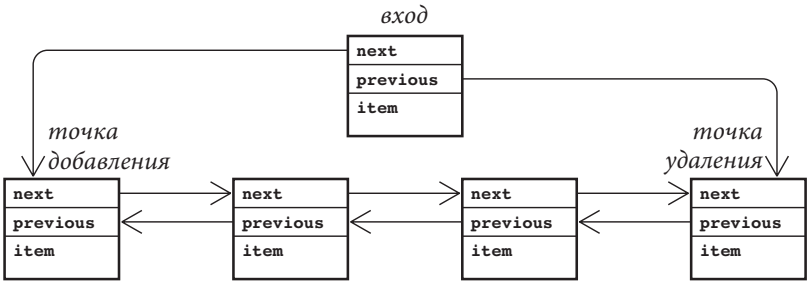

Рис. 1.4. Очередь в виде двусвязного списка из четырех элементов

Реализация очереди в виде двусвязного списка выглядит так:

```
typedef struct qu_t { item_t item; struct qu_t *next;
                      struct qu_t *previous; } queue_t;
queue_t *create_queue()
{ queue_t *entrypoint;
  entrypoint = (queue_t *) malloc(sizeof(queue_t));
  entrypoint->next = entrypoint;
  entrypoint->previous = entrypoint; return(entrypoint);
}

int queue_empty(queue_t *qu)
{ return(qu->next == qu);
}

void enqueue(item_t x, queue_t *qu)
{ queue_t *new;
  new = get_node(); new->item = x;
  new->next = qu->next; qu->next = new;
  new->next->previous = new; new->previous = qu;
}

item_t dequeue(queue_t *qu)
{ queue_t *tmp; item_t tmp_item;
  tmp = qu->previous;
  tmp_item = tmp->item;
  tmp->previous->next = qu;
  qu->previous = tmp->previous;
  return_node(tmp);
  return(tmp_item);
}

item_t front_element(queue_t  *qu)
{ return(qu->previous->item);
```

```
}

void remove_queue(queue_t *qu)
{ queue_t *tmp;
  qu->previous->next = NULL;
  do
  { tmp = qu->next; return_node(qu);
    qu = tmp;
  }
  while( qu != NULL );
}
```

Какая из реализаций лучше – дело вкуса. Обе они несколько сложнее стека, хотя сами структуры выглядят почти одинаково.

Как и стек, очередь – это динамическая структура данных с операциями изменения очереди `enqueue` и `dequeue` и обращения к очереди `queue_empty` и `front_element`, каждая из которых имеет постоянное время выполнения. Кроме них, есть еще операции `create_queue` (создать очередь) и `delete_queue` (удалить очередь) с теми же ограничениями, что и у аналогичных операций для стека: создание очереди в массиве требует от системы управления памятью выделения большого блока памяти, тогда как для создания очереди в виде списка требуется всего лишь несколько элементов. Удаление очереди в массиве – это просто возврат блока в системную память, тогда как удаление очереди в виде списка требует возврата в системную память каждого его элемента. Для удаления очереди в виде списка из n элементов требуется время $O(n)$.

1.3. Двусторонняя очередь

Двусторонняя (double-ended) очередь – это очевидное обобщение стека и очереди. Это очередь, где добавление и удаление могут выполняться с обеих ее сторон. Реализовать ее можно либо в массиве, либо, как обычную очередь, в двусвязном списке. Поскольку в ее реализации нет ничего нового, ее код здесь не приводится. Двусторонняя очередь применяется редко, а вот «полуторасторонняя очередь» («one-and-a-half ended queue») иногда полезна, как в случае с `minqueue`, описанной в разделе 5.11.

1.4. Динамическое выделение памяти для элементов

В предыдущих разделах для динамического создания и удаления элементов постоянного размера использовались операции `get_node` и `return_node`, которые отличаются от системных операций `malloc` и `free`, применяющихся только для объектов памяти произвольного, обычно большого, размера. Суть этого различия в том, что в конечном счете процесс выделения памяти – это единственный способ получить необходимый ресурс, и этот про-

цесс довольно сложен, так что нет гарантии, что время его выполнения будет постоянным. В любой эффективной реализации динамической структуры, где ее элементы постоянно появляются и исчезают, нельзя позволять каждой операции опускаться до низкоуровневого управления памятью операционной системы. С этой целью вводится некий промежуточный уровень, получающий доступ к низкоуровневому управлению памятью лишь изредка, когда нужно выделить большой блок памяти, который возвращается обратно в систему лишь небольшими порциями постоянного размера, то есть поэлементно.

Вообще говоря, эффективность операций `get_node` и `return_node` имеет решающее значение для любой динамической структуры, но, к счастью, нам не нужно создавать свою систему управления памятью, так как у нас есть два существенных упрощения. В отличие от функции `malloc`, выделяющей блоки памяти произвольного размера, мы имеем дело только с объектами постоянного размера и до завершения программы можем не возвращать память из промежуточного уровня в системный. И это разумно: выделенная на промежуточном уровне структуре данных память достаточна на данный момент, неизменна в объеме и не должна незамедлительно освобождаться для других параллельных программ или структур.

В таком случае для *повторного* динамического выделения памяти элементам можно использовать *свободный список* (free list), содержащий не используемые в данный момент элементы. При каждом удалении элемента он просто добавляется к этому списку. Для операции `get_node` ситуация усложняется: если свободный список не пустой, то элемент берется из него; если же он пустой, а в текущем блоке памяти еще есть свободное место, то новый элемент создается в этом блоке. В противном случае с помощью функции `malloc` нужно выделить новый блок памяти и создать в нем новый элемент.

Реализация такой схемы может выглядеть следующим образом:

```
typedef struct nd_t { struct nd_t *next;
                      /*и другие поля*/ } node_t;
#define BLOCKSIZE 256
node_t *currentblock = NULL;
int size_left;
node_t *free_list = NULL;

node_t *get_node()
{ node_t *tmp;
  if( free_list != NULL )
  { tmp = free_list;
    free_list = free_list->next;
  }
  else
  { if( currentblock == NULL || size_left == 0 )
```

```
    { currentblock = (node_t *) malloc(BLOCKSIZE * sizeof(node_t));
      size_left = BLOCKSIZE;
    }
    tmp = currentblock++;
    size_left -= 1;
  }
  return(tmp);
}

void return_node(node_t *node)
{ node->next = free_list;
  free_list = node;
}
```

Обычно динамическое выделение памяти – это источник множества ошибок, с трудом поддающихся исправлению. Простая дополнительная предосторожность во избежание некоторых распространенных ошибок – добавить к элементу структуры еще одно поле `int_valid` и заполнять его различными значениями в зависимости от того, был ли он только что возвращен операцией `return_node` или выделен операцией `get_node`. В этом случае можно удостовериться, что указатель действительно ссылается на существующий элемент и все ранее возвращенные `return_node` элементы тоже существуют.

1.5. Теневые копии структур в массиве

Простота таких структур позволяет обойти ограничение их предельного размера в массиве. Для этого одновременно поддерживаются две копии структуры – активная (текущая) структура и ее копия большего размера, которая создается так, чтобы она была вполне готова к работе с ней до достижения активной структурой своего максимального размера. Для этого каждая операция со структурой, помимо прочего, копирует фиксированное количество элементов из старой структуры в новую. Как только старая структура полностью скопирована в новую более крупную, старая структура удаляется, а новая становится активной, и по мере необходимости создается еще более крупная копия. Кажется, что это просто и порождает лишь издержки преобразования структуры фиксированного размера в неограниченную структуру. Но есть некоторые нюансы: активная структура изменяется во время копирования, и эти изменения должны быть учтены в еще неполной большей копии. Для демонстрации этого принципа приведем код для стека в массиве:

```
typedef struct { item_t *base;
                 int size;
                 int max_size;
```

```
                item_t *copy;
                int copy_size; } stack_t;

stack_t *create_stack(int size)
{ stack_t *st;
  st = (stack_t *) malloc( sizeof(stack_t));
  st->base = (item_t *) malloc(size * sizeof(item_t));
  st->max_size = size;
  st->size = 0;
  st->copy = NULL;
  st->copy_size = 0;
  return(st);
}

int stack_empty(stack_t *st)
{ return(st->size == 0);
}

void push(item_t x, stack_t *st)
{ *(st->base + st->size) = x;
  st->size += 1;
  if( st->copy != NULL || st->size >= 0.75*st->max_size )
  { /* продолжить или начать копирование */
    int additional_copies = 4;
    if( st->copy == NULL )
    /* начать копирование: выделить область */
    { st->copy =
            (item_t *) malloc(2 * st->max_size * sizeof(item_t));
    }
    /* продолжить копирование хотя бы 4 элементов */
    while( additional_copies > 0 && st->copy_size < st->size )
    { *(st->copy + st->copy_size) = *(st->base + st->copy_size);
         st->copy_size += 1;
         additional_copies -= 1;
    }
    if( st->copy_size == st->size ) /* копия готова */
    { free(st->base);
      st->base = st->copy;
      st->max_size *= 2;
      st->copy = NULL;
      st->copy_size = 0;
    }
  }
}
```

```
item_t pop(stack_t *st)
{ item_t tmp_item;
  st->size -= 1;
  tmp_item = *(st->base + st->size);
  if( st->copy_size == st->size ) /* копия готова */
  { free(st->base);
    st->base = st-> copy;
    st->max_size *= 2;
    st->copy = NULL;
    st->copy_size = 0;
  }
  return(tmp_item);
}

item_t top_element(stack_t *st)
{ return(*(st->base + st->size - 1));
}

void remove_stack(stack_t *st)
{ free(st->base );
  if( st->copy != NULL )
     free(st->copy);
     free(st);
}
```

Для стека ситуация особенно проста, поскольку сводится лишь к его копированию от основания до текущей вершины, так как между ними ничего не меняется. Пороговый размер активной структуры для запуска копирования (здесь `0,75*size`), максимальный размер новой структуры (здесь вдвое больший старой) и количество копируемых на каждом шаге элементов (здесь 4), конечно, должны быть выбраны так, чтобы копирование закончилось до переполнения старой структуры. Заметьте, что копирование может закончиться в двух случаях – при фактическом копировании в `push` и при удалении еще не скопированных элементов в `pop`.

В самом общем случае связь между пороговым размером, максимальным новым размером и количеством копируемых элементов выглядит следующим образом:

- если активная структура имеет максимальный размер s_{max},
- и копирование начинается по достижении αs_{max} (где $\alpha \geq ½$),
- и новая структура имеет максимальный размер $2s_{max}$,
- и каждая операция увеличивает фактический размер не более чем на 1,

то остается не менее $(1 - \alpha)s_{max}$ шагов для завершения копирования не более s_{max} элементов из активной структуры в новую бóльшую структуру.

Поэтому, чтобы закончить копирование до переполнения активной структуры, нужно скопировать в каждой операции $\lceil 1/(1 - \alpha) \rceil$ элементов. При создании новой структуры ее размер удваивается, хотя можно было бы выбрать и другой размер βs_{max}, где $\beta > 1$, при условии что $\alpha\beta > 1$. Иначе процесс копирования пришлось бы начинать заново – до завершения предыдущего.

В принципе, этот метод является довольно общим и применим не только к структурам в массивах. Он еще будет использоваться в разделах 3.6 и 7.1. Всегда есть возможность преодолеть фиксированный размер структуры, скопировав ее содержимое в бóльшую структуру. Но не всегда ясно, как разбить процесс копирования на ряд небольших шагов, которые могут выполняться параллельно с обычными операциями над структурой, как в нашем примере. Вместо этого можно скопировать всю структуру за один шаг и оценивать не худшее время выполнения, а среднее.

Последний пример применения такого подхода и связанных с ним проблем – это реализация расширяемого массива. Обычные массивы имеют фиксированный размер; им отводится определенная область памяти, которая не может увеличиваться во избежание возможных конфликтов с уже выделенными для других переменных областями. Доступ к элементу такого массива довольно быстр, и требуется всего лишь одно вычисление адреса. Но некоторые системы поддерживают массивы, размер которых может расти. Доступ к элементу такого массива гораздо сложнее и должен поддерживать следующие операции:

- `create_array` – создать массив заданного размера;
- `set_value` – присвоить значение элементу массива с заданным индексом;
- `get_value` – получить значение элемента массива с заданным индексом;
- `extend_array` – увеличить длину массива.

Для реализации такой структуры используется тот же метод создания теневых копий. Но здесь появляется новая проблема, поскольку моделируемая структура растет при каждой операции не на один элемент. Например, при выполнении операции `extend_array` их может быть гораздо больше. Но тем не менее можно легко добиться постоянного среднего времени выполнения одной и той же операции.

При создании массива размера s ему выделяется область, бóльшая, чем требуется. Допустив, что размер выделяемых массивов всегда кратен степени двух, можно выделить ему сначала область размера $2^{\lceil \log s \rceil}$ и сохранить в заголовке этой структуры ссылку на его начальную позицию, а также его текущий и максимальный размеры. Таким образом, обращение к элементам массива всегда будет начинаться с обращения к его заголовку. При каждом выполнении операции `extend_array` прежде всего проверяется, превышает ли его текущий (максимальный) размер требуемый. Если

это так, то просто увеличивается текущий размер. В противном случае выделяется новый массив, размер которого на 2^k больше требуемого, и каждый элемент старого массива копируется в новый. Таким образом, обращение к элементу массива всегда требует постоянного времени $O(1)$, то есть одного перехода по ссылке. А вот расширение массива может потребовать уже линейного времени, зависящего от размера массива. Но средняя сложность при этом не так уж плоха. Если предельный размер массива равен $2^{\lceil \log k \rceil}$, то в худшем случае в `extend_array` на операции копирования массивов размером 1, 2, 4, ..., $2^{\lceil \log k \rceil - 1}$ будет затрачено в общей сложности время $O(1 + 2 + \ldots + 2^{\lceil \log k \rceil - 1}) = O(k)$ для каждой операции `extend_array` без копирования массива. Следовательно, мы имеем следующую сложность:

> **Теорема.** В структуре расширяемого массива с теневыми копиями любая последовательность из n операций `set_value`, `get_value` и `extend_array` для массива с предельным размером k выполняется за время $O(n + k)$.

Если допустить, что обращение к элементам массива выполняется только один раз, то предельный размер – это не более чем количество обращений к элементам массива, что дает среднюю временну́ю сложность $O(1)$ на одну операцию.

Конечно, было бы лучше разнести копирование элементов массива по более поздним операциям обращения к его элементам, но у нас нет возможности контролировать операцию `extend_array`. Очередное расширение массива возможно до завершения копирования текущего массива, поэтому наш метод не сработает для такой структуры. Еще одна существенная проблема с расширяемыми массивами состоит в том, что указатели на элементы массива отличаются от обычных указателей, поскольку положение массива в памяти может меняться. Поэтому в самом общем случае следует избегать расширяемых массивов, даже если язык программирования их поддерживает. Другой способ реализации расширяемых массивов рассматривался в [129].